The IA-64 Architecture at Work

Two key architectural features—predication and control speculation—will enable IA-64 compilers to extract instruction-level parallelism. To show how compilers will use IA-64 instructions, the author uses code fragments from the pointer-chasing problem—an inherently serial code—and from a nested loop with difficult-to-predict branches.

Carole Dulong
Intel Corp.

Over the past several years, strategies to increase microprocessor performance have focused on finding more *instruction-level parallelism*. ILP is basically the idea of finding several instructions to execute at the same time. By providing multiple functional units on which to execute instructions, computer architects expect to improve performance.

However, two difficult problems limit ILP:

- *branch instructions*, which introduce control dependencies, and
- *memory latency*, the time it takes to retrieve data from memory.

In the absence of new programming languages that are explicitly parallel, the task of "exposing" ILP falls to the compiler. In IA-64, Intel's upcoming 64-bit architecture (see the "IA-64 to Date" sidebar for current information), the compiler will play a pivotal role in using predication and control speculation to expose more ILP.

Two code fragments are used to illustrate predication and control speculation. The fragments are scheduled with actual IA-64 instructions and are representative of general-purpose integer code, such as that found in computer-aided design and database applications. A comparison of performance with and without the two features demonstrates how predication and control speculation can reduce the number of cycles required to execute an instruction and improve performance.

PREDICATION

The IA-64 architecture uses a *full predication* model, in which a compiler can append a predicate to all instructions. *Predicates* are simply tags that permit a program to execute instructions conditionally, depending on the predicate's value, which in turn depends on the outcome of a conditional statement. An instruction with a predicate value of true executes normally. If the predicate is false, the associated instruction—although issued—does not write its results to registers or memory. Research has shown predication to be effective at removing branches and at decreasing penalties from branch mispredicts.[1] A simple code example with a difficult-to-predict branch illustrates how predication can remove the branch.

Figure 1a shows the C code for a classic if-then-else statement. In a traditional architecture, the processor loads the data from memory, compares the value of `a(i).ptr` with zero, and uses the compare's (`cmp`'s) result in a conditional-branch instruction. Because of the conditional branch, a traditional compiler structures this code into four basic blocks, as shown in Figure 1b. The processor must execute the instructions of all four blocks serially, and branch instructions are barriers to ILP. Predication is used to remove the difficult-to-predict branch in the first basic block.

In the IA-64 architecture, compare instructions generate two predicates, as shown in Figure 1c:

0018-9162/98/$10.00 © 1998 IEEE

IA-64 to Date

The IA-64 architecture is Intel's first 64-bit architecture. The company plans to release information on IA-64 incrementally; this is a summary of official information to date.

In developing the IA-64 architecture, Intel sought to provide

- full binary compatibility with IA-32 software,
- scalability over a wide range of implementations with headroom to meet future requirements, and
- full 64-bit computing.

IA-64 is based on the EPIC (Explicitly Parallel Instruction Computing)[1] design philosophy, which has three components:

- *Explicit parallelism*. An EPIC-based architecture makes ILP explicit in the machine code. The compiler scheduling scope is inherently larger than a hardware-scheduling window (or reorder buffer). It is thus more efficient to let the compiler find the ILP than to have the hardware discover the ILP in serial machine code.
- *Features to enhance ILP.* IA-64 provides features that help the compiler expose and express ILP. Two of these features are predication and control speculation, the focus of this article.
- *Resources for parallel execution*. The IA-64 architecture provides many more registers than current commercial processors: 128 integer, 128 floating-point, and 64 predicate. A processor requires many registers to efficiently use multiple functional units.

Figure A1 is a graphical representation of how IA-64 depends on the compiler to parallelize code to run on multiple hardware resources (functional units).

Figure A2 shows the IA-64 instruction bundle, which contains three, 40-bit IA-64 instructions. The template is a means of expressing explicit instruction dependencies while also allowing the compiler to flexibly group several independent instructions across multiple bundles.

Reference

1. L. Gwennap, "Intel, HP Make EPIC Disclosure," *Microprocessor Report*, Oct. 27, 1997, pp. 1, 6-9.

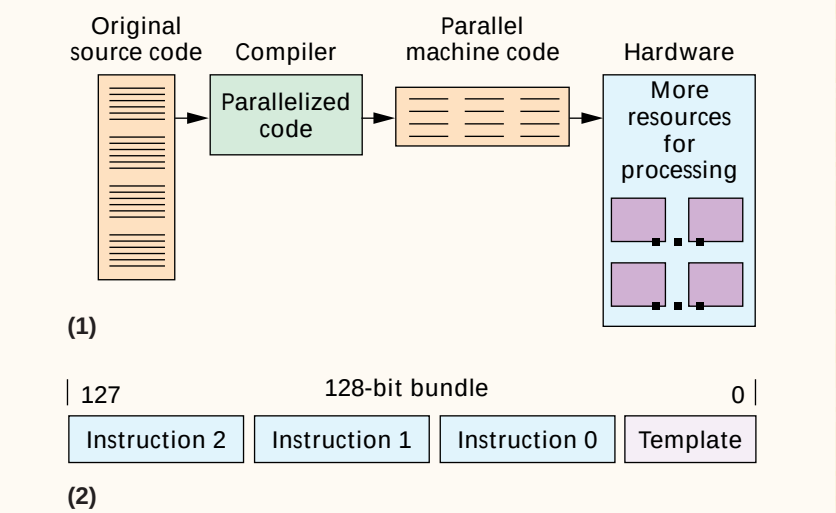


Figure A. (1) A graphical representation of the basic concepts behind explicit parallelism and (2) the 128-bit bundle of three IA-64 instructions with a template.

- a true predicate that is set to 1 if the result of the compare is true, and
- a false predicate that is set to 1 if the result of the compare is false.

The `then` path executes if the result of the compare is true, so the true predicate, p1, is used to predicate this path. The `else` path executes if the result of the compare is false, so the false predicate, p2, is used to predicate the `else` path.

The resulting code has no branch, so the `then` and the `else` path can execute in parallel. Since p1 and p2 are mutually exclusive (only one can be true at a given time), the two store instructions can execute in parallel, even though they store their results at the same address. The predicates permit only one of the two stores to write its result to memory. The result of the other store (the one with a false predicate) is thrown away. In this way, predication exposes instruction-level parallelism by removing branches and their associated mispredict penalties. An IA-64 architecture can thus schedule an if-then-else statement in one basic block, as shown in Figure 1c.

CONTROL SPECULATION

Allowing the compiler to speculatively execute load instructions effectively minimizes the impact of memory latency and increases instruction-level parallelism.[2] The IA-64 architecture permits *hoisting*, a way of moving a load instruction's execution to precede its controlling branch. Loading data before the program needs it hides memory latency. It also permits a processor with multiple execution units to run more instructions in parallel.

Figure 2a shows a code fragment that illustrates control speculation. Let us assume that array `[a]` is large and does not fit in the cache. Thus it is important for the compiler to load the array elements from memory as early as possible to hide the memory latency. In this example, hoisting the instruction that loads `a[t1-t2]` to a point before the branch is not possible unless the architecture provides support for speculation of any load, regardless of the safety of its address. For example, if `t1` is not greater than `t2`, the address `a[t1-t2]` is invalid, and loading from an invalid address would cause an exception.

The IA-64 architecture introduces a new instruc-

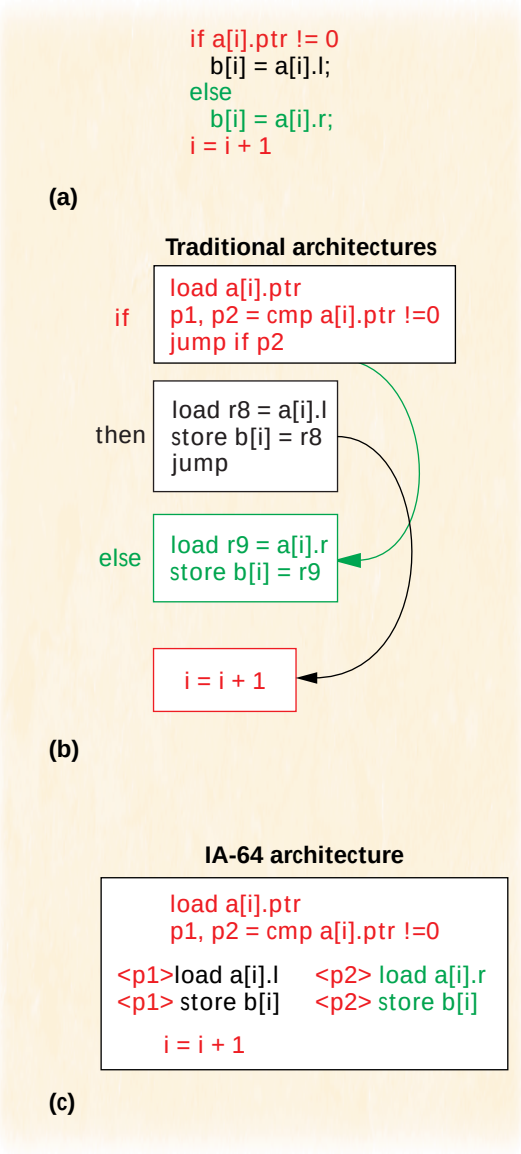


Figure 1. Classic if-then-else statement in (a) C code becomes (b) four basic blocks in a traditional processor architecture. In the IA-64 architecture, the same loop executes in (c) one basic block.

tion called *speculative load* (`ld.s`), which executes the memory fetch, performs exception detection, but does not deliver the exception (does not call the OS routine that handles the exception). An instruction called `check.s` remains in the load's home block and delivers the exceptions. If `ld.s` detects an exception, the target register is marked by setting a token bit. The `check.s` instruction for this register branches to fix-up code if the token bit is set.

The speculative-load mechanism allows load instructions to execute before the program determines whether the data addresses are valid. In this example, a speculative load hoists the load of `a[t1-t2]` above the instruction that compares `t1` with `t2`. If `t1` is less than or equal to `t2`, the token bit of register `r8` is set, the program jumps out of this path, and the exception detected by this load never executes. If `t1` is greater than `t2`, then the load instruction completes normally, and the processor executes the load instruction's home block. The `check.s r8` instruction executes and does nothing if the `r8` token bit is not set. If address `a[t1 -t2]` is valid but the load instruction encounters a page-fault exception, for example, the token bit of `r8` is set. When the `check.s r8` instruction detects the set token bit, program control jumps to fix-up code that brings the page in, executes the load, and restarts execution after `check.s`.

THE IA-64 ARCHITECTURE AT WORK

The following examples illustrate the use of predication and control speculation in actual code fragments that are representative of general-purpose integer code. The first fragment comes from li (for Lisp interpreter), the SPECint benchmark. This fragment, called `xlxgetvalue`, illustrates the use of control speculation to hide memory latency and to load data before determining pointer validity.

The second example is a code fragment called `treeins`, which inserts a new element into a tree data

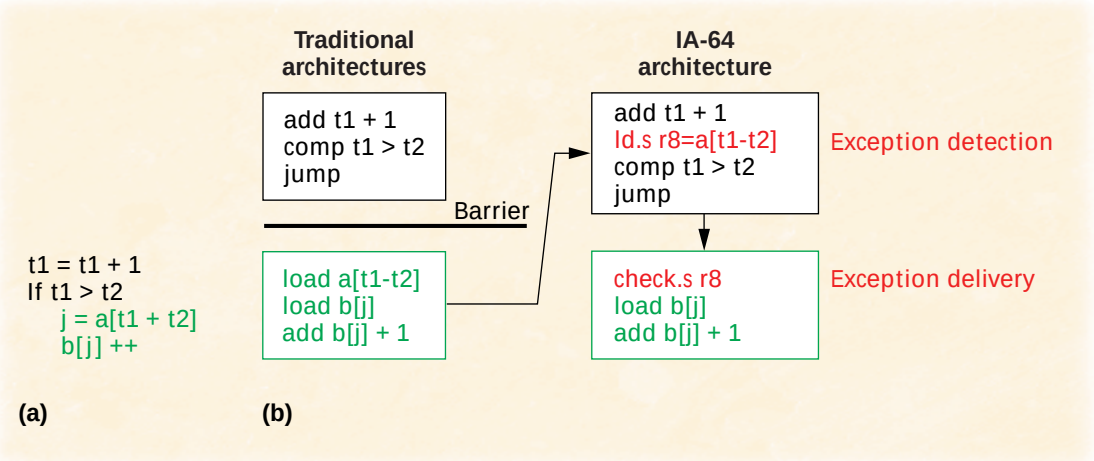


Figure 2. Example that illustrates control speculation: (a) code fragment and (b) control flow diagrams for traditional and IA-64 architectures. Arrow shows the hoisting of a load to a point before its branch is resolved.

structure. It illustrates the use of predication and shows how to schedule all paths of a double-nested if-then-else statement to execute in parallel. Both examples are code fragments that display very little instruction-level parallelism in traditional architectures and are also highly serial.

Xlxgetvalue

Xlxgetvalue illustrates a classic problem—walking through linked lists—known as the *pointer-chasing* problem. Control speculation enables IA-64 to expose the instruction-level parallelism in this inherently serial code.

Figure 3 shows the sample C code and its corresponding control flow diagram. The loop of interest is the inner loop, colored green. Induction variable `ep` is dereferenced (used as a pointer to data) twice and compared to the value `sym`. Once the processor finds `sym`, control flow exits the loop. The two loads of `x = car(ep)`, and `y = car(x)` are in the loop's critical path. To minimize this critical path, it is important to start the loads as early as possible. Control speculation allows the compiler to schedule the load as soon as `ep` is known, but before the processor determines whether it is a valid pointer.

Figure 4 shows the machine configuration used in the sample schedule for `xlxgetvalue`. Note that this configuration is only a hypothetical example and does not correspond to any real machine. It assumes that there are six functional units and that all the units can execute all the instructions (except for loads and stores). Only two units have access to the level-0 data cache, which allows those two units to issue load and store instructions. This model also assumes that the level-0-cache latency is one clock.

It is important to keep in mind that code optimized for this machine configuration would still function on any other machine configuration. The IA-64 architecture guarantees compatibility across machines that have different numbers of functional units or different latencies for functional units. IA-64, unlike traditional VLIW architectures,

- does not expose latencies (the hardware does not rely on the compiler to schedule for correct latencies) and
- provides hardware that is fully interlocked (has scoreboards to track latencies).

However, very much like traditional architectures, EPIC architectures achieve their highest performance on a given machine when programs are compiled for a machine-specific configuration.

First iteration. Figure 5 shows a schedule for the first iteration of `xlxgetvalue`. The C statements above the schedule correspond to the machine-level

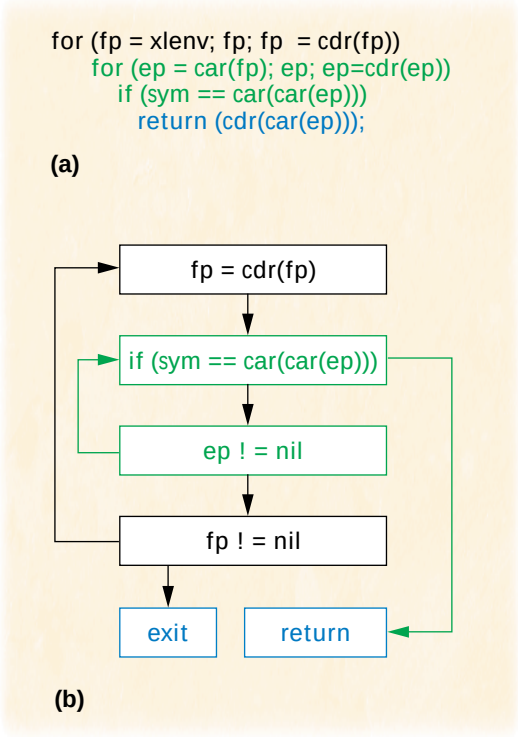


Figure 3. Xlxgetvalue (a) C code fragment and (b) control flow diagram.

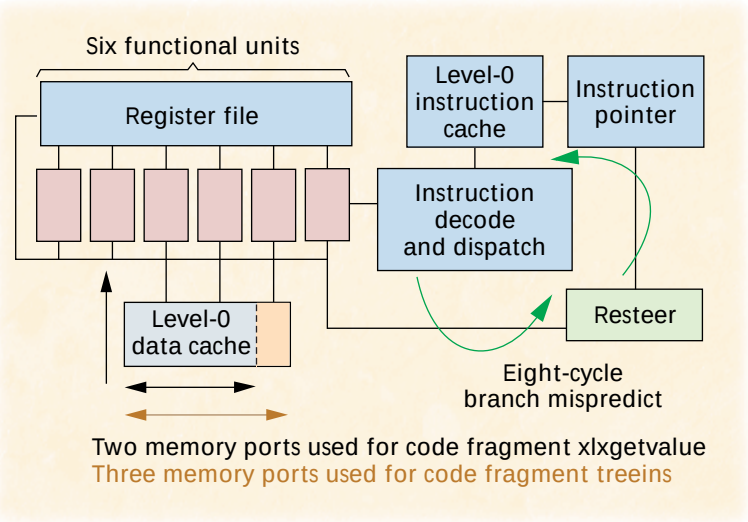


Figure 4. Hypothetical machine configuration used to schedule both code fragment examples.

instructions of the same color. Columns represent the six functional units of the hypothetical machine configuration. Rows represent the instructions that can execute in parallel in a given cycle. For example, the instruction `ld ep1` issues in cycle one. Assuming the cache latency is one clock, the processor can use the load's result in cycle two. This is why the `cmp` instruction that generates `cond1` issues in cycle two.

In Figure 5, speculative-load `ld.s` occurs in cycle two. This instruction accesses `car(ep1)` before the processor determines whether or not `ep1` is a valid

Figure 5. Schedule of the first iteration of xlxgetvalue.

for (ep = car(fp); ep; ep = cdr(ep))
if (sym == car(car(ep)))

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Ld ep1					
2	Cond1 = ep1 == nil	Ld.s car(ep1)	Br nxt_fp if cond1			
3	Check.s	Ld x = car (car(ep1))				
4	Cond2 = sym == x	Br return if cond2	Br nxt_ep			

Figure 6. Schedule for two iterations of xlxgetvalue.

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
0	Ld ep1					
1	Ld.s car(ep1)	Cond1 = cmp ep == nil	Ld.s ep2 = cdr(ep1)	Br nxt_fp if cond1		
2	Check.s	Ld car car(ep1)	Ld.s car(ep2)	Cond3 = cmp ep2==nil		
3	Cond2 = cmp == sym	Ld.s car car(ep2)	Br return if cond2	Br nxt_fp if cond3		
4	Check.s	Ld nxt ep1 = cdr(ep2)	Cond4 = cmp == sym	Br return if cond4	Br nxt_ep	

pointer. The validity of `ep1` is not determined before cycle three, when the check corresponding to the instruction `ld.s car(ep1)` can execute. Once the processor loads `car(ep1)` in cycle two, it can load `car(car(ep1))` in cycle three. Finally, the processor can use the result of this second load in cycle four to compare against `sym`. If the sought-for value is found, the loop exits. If the `compare` is not true, the loop branches to the next iteration in the schedule, `nxt_ep`. This schedule shows that one loop iteration can execute in four clocks.

If the speculative load of `car(ep1)` in cycle two generates a page fault, for example, the page-fault exception would not be delivered before the execution of `check.s` in cycle three. The check would then branch to fix-up code, which would bring in the page and reexecute `load car(ep1)`. Normal execution could then resume at the first instruction after the check, which is the load of `car(car(ep1))`. If `ep1` was the nil pointer, execution of this loop iteration would stop at cycle two, and the code would branch to the next outer-loop iteration (called `nxt_fp` in Figure 5). In this case, the exception created by the speculative load would not be delivered to the program because it does not need the load's result.

Note that the compiler has scheduled two branch instructions in cycle four. If the first branch is taken (`return if cond2`), the second branch does not execute. If the first branch is not taken, then the second branch—which is unconditional in this example—is taken.

It is obvious that one iteration of this loop does not use all the machine resources and leaves many units available. The next step is to unroll the loop to expose instruction-level parallelism and to take advantage of the machine's multiple execution units.

Unrolling the loop. Figure 6 shows the schedule for the `xlxgetvalue` code fragment with the inner loop unrolled twice. The instructions corresponding to the first iteration are the same as shown in Figure 5; they

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1		Cond1 = cmp ep == nil		Br nxt_fp if cond1		
2	Ld car(ep1)	Ld ep2 = cdr(ep1)				
3	Ld car car(ep1)		Cond3 = cmp ep2==nil			
4	Cond2 = cmp == sym	Ld car(ep2)	Br return if cond2	Br nxt_fp if cond3		
5		Ld car car(ep2)				
6	Ld nxt ep1 = cdr(ep2)	Cond4 = cmp == sym	Br return if cond4	Br nxt_ep		

Figure 7. Schedule of xlxgetvalue without control speculation.

appear in black. The machine instructions that correspond to the second iteration use the same color coding as that of the C code in the upper right.

The loop's second iteration provides new examples of control speculation's use. In cycle one, the processor loads the next value of the induction variable, `ep2`, speculatively—that is, before it determines if the first iteration's induction variable is valid. The two loads thus depend on `ep2` as well as the speculative loads of `car(ep2)` in cycle two and of `car(car(ep2))` in cycle three. The program branches out of the sequence at cycle three if `ep2` is the nil pointer (by taking the branch to `nxt_fp` if `cond3` is true), so all instructions using `ep2` before that point are speculative.

All instructions propagate token-bit values associated with all registers. In other words, if any operand of any instruction has its token-bit set, the token bit of the result register is also set to 1. In this example, if any speculative load triggers an exception, then the processor will set the token bit of the result register of the last speculative load. This means that the program uses only one `check.s` instruction for the chain of three dependent loads. Checking the token bit of the last register loaded in the chain will deliver an exception to `check.s` if any load instruction creates an exception.

Fix-up code would test all registers loaded by the chain of dependent load instructions, and it would

- bring in the missing page in the case of a page fault and
- reexecute all the load instructions not executed in the first pass.

Since the machine configuration has a two-port data cache, this processor can execute two loads in parallel in cycles one and two. The schedule for the second iteration is very similar to the schedule shown in Figure 5 for the first iteration. Figure 6 shows that the compiler can schedule both iterations in parallel in four clocks. The load of induction variable `ep1` (`Load next ep1`) can occur at the end of the loop in cycle four. The load of `ep1` shown at cycle 0 is actually executed only one time when entering the loop and does not belong to the inner loop.

Without control speculation, the static schedule of this code fragment would take six clocks for two iterations, as shown in Figure 7, instead of four clocks. The loads dereferencing induction variable `ep1` cannot start before `ep1` has been checked against the nil pointer (the two loads circled with red in Figure 7). Similarly, the load dereferencing `ep2` cannot execute before the processor checks `ep2` against the nil pointer. This load executes in cycle four even though `ep2` is available in cycle three.

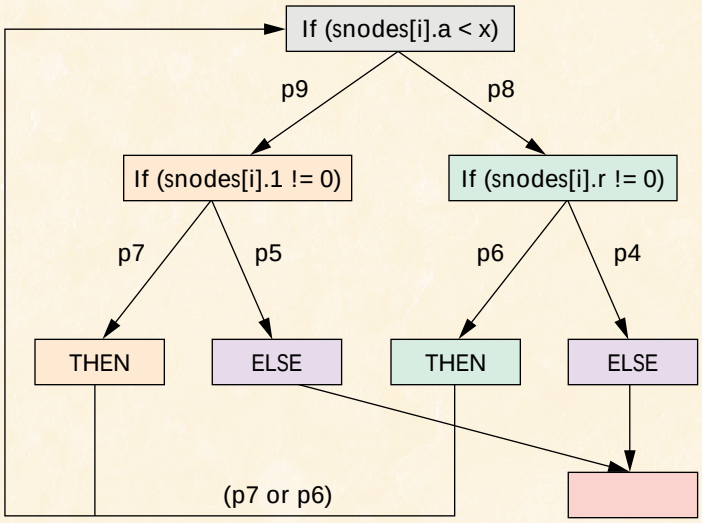
With control speculation, the `xlxgetvalue` loop executes in two clocks per iteration; without control speculation, it executes in three clocks per iteration in this example machine configuration. A one-cycle difference can add a significant performance benefit in loops that execute millions of times in the overall li benchmark.

Treeins code fragment

This code fragment illustrates how predication removes difficult-to-predict branches and exposes instruction-level parallelism. Figure 8 shows the C code and corresponding control flow diagram. The task is to insert value `x` into a treelike data structure. At each point in the tree, the program compares `x` with existing data value `a` to determine whether `x` belongs in the right (`s.node.r`) or left (`s.node.l`) side of the tree structure. The program tests each node, and only inserts the new element `x` at a leaf point in the tree. It searches the tree until it finds a null node.

```
L10: /* compare */
     if (s.nodes[i].a < x)
       if (s.nodes[i].l != 0) {
         i = s.nodes[i]. l;
         goto L10;}
       else {
         s.nodes[i].l = j;
         goto L20;}
     else{
       if (s.nodes[i].r != 0){
         i = s.nodes[i].r;
         goto L10;}
       else{
         s.nodes[i].r = j;
         goto L20;}
     }
L20: /* insert */
       s.nodes[j].a = x;
       s.nodes[j].l = 0;
       s.nodes[j].r = 0;
       s.nxt_avail = j+1;
```

(a)



(b)

Figure 8. Treeins (a) C code fragment and (b) control flow diagram.

For this code fragment, the schedule assumes a machine configuration very similar to that of the previous example. The difference between the two models is the number of data cache ports. This machine configuration has three ports on the data cache, so it can execute up to three load or store instructions every cycle.

Then paths. For this code example, the compiler compiles the `then` paths first. These paths are highlighted by the arrows in Figure 8. The color coding for the different paths in Figure 8 corresponds to that for the C code and the machine code in the following figures.

Figure 9 shows a schedule with both `then` paths scheduled in parallel, without lengthening the cycle count of either path. The C code for the two `then` paths is also shown.

First, the program must compute the address of `s.nodes[i].a`. The s.nodes data structures have three integer elements (each four bytes wide); thus, induction variable *i* is multiplied by 12. The program accomplishes this with two shift-left-add instructions—one shift-left-add by two (to multiply by four) and one shift-left-add by three (to multiply by eight).

A displacement must be added to get to the address of the data structure's element `a`. This is why there are two `sh1add` instructions and one add instruction before the load of `s.nodes[i].a`. Since this machine configuration has three memory ports, the loads of all three elements of `s.nodes[i]` can execute in parallel in cycle four. The processor computes predicates in cycle five. P9 is true if `a` is less than *x*. P8 is true if `a` is greater than or equal to *x*. Predicates p9 and p8 are used in cycle six to predicate the compare instructions for the `r` and `1` values. The compare of the `s.nodes[i].1` value computes two complementary predicates, p7 and p5. The compare of the `s.nodes[i].r` value computes the two complementary predicates, p6 and p4.

When the qualifying predicate of a compare instruction (p9 or p8, in this case) is false, both result predicates (p5 and p7, or p4 and p6) are also false.

Predicates computed in a given cycle can be used in the same cycle by branch instructions. This is why the two branch instructions predicated by p6 and p7 can execute in cycle six. The two predicated move instructions in cycle five copy into the register holding *i* for either the `1` or `r` side. Predicates p9 and p8 are complementary, so only one instruction writes its result.

Else paths. The schedule accommodates the two `then` paths in six clocks. Next, the compiler schedules the instructions for the remaining `else` paths in parallel with the paths already scheduled. Figure 10 shows the overall schedule for the treeins code fragment. The C code for the `else` paths uses the same color coding as that for the corresponding machine instructions.

The processor calculates the address of `s.nodes[j]` in cycle one, two, and three the same way it calculates the address of `s.nodes[i]`—with two shift-left-add instructions and one add instruction.

Cycle seven can accommodate the two store instructions that correspond to the `else` statements. The store instructions are predicated by complementary predicates p4 and p5, and only one store writes its result in memory.

The compiler then schedules the four store instruc-

```
if (snodes[i].a < x)          else{
    if (snodes[i].l != 0) {       if (s.nodes[i].r != 0) {
      i = snodes[i].l;              i = s.nodes[i].r;
      goto L10;}                    goto L10;}
```

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Shladd					
2	Shladd					
3	Add	Add				
4	Ld a	Ld l	Ld r			
5	Cmp p9, p8= a<x					
6	<p9> Cmp p7, p5= l!=0	<p8> Cmp p6, p4 r!=0	<p9> Mov i=l	<p8> Mov i=r	<p6> Br nxt_loop	<p7> Br nxt_loop

Figure 9. Schedule of just the then paths from treeins.

```
else{                  L20: /* insert */          else{
    s.nodes[i]. = j;           s.nodes[j].a = x;        s.nodes[i].r = j;
    goto L20;}                 s.nodes[j].l = 0;        goto L20;}
                               s.nodes[j].r = 0;
                               s.nxt_avail = j+1;
```

Cycle	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
1	Shladd		Shladd			
2	Shladd		Shladd			
3	Add	Add	Add	Add	Add	Add
4	Ld a	Ld l	Ld r			
5	Cmp p9, p8= a<x					
6	<p9>Cmp p7, p5= l!=0	<p8> Cmp p6, p4 r!=0	<p9> Mov i=l	<p8> Mov i=r	<p7> Br nxt_loop	<p6> Br nxt_loop
7	<p5> Store [i].l	<p4> Store [i].r	Store [j].a			
8	Store [j].l	Store [j].r	Store nxt_avail			

Figure 10. Final and complete schedule for treeins.

tions in the common path of both `else` statements in cycle seven and cycle eight. All stores could occur in parallel, but the three available cache ports limit the schedule. These stores do not need to be predicated since they execute along both `else` paths. If the program follows the `then` paths, one of the branch instructions executed in cycle six is taken, and the instructions in cycles seven and eight do not execute.

Final schedule. The final, complete schedule shown in Figure 10 fits the `then` paths into six clocks, and the

`else` paths into eight clocks. This is an average of less than seven clocks per iteration, since the `then` paths will likely execute many times before the loop exits through the `else` paths. Note that the compiler schedules all four paths (the two `then` paths and the two `else` paths) in parallel, which exposes a significant amount of instruction-level parallelism.

Without predication, the schedule for this code fragment would include two branch instructions that

- are difficult to predict (the branch misprediction rate is actually above 25 percent for both branches) and
- incur high mispredict penalties (the example machine configuration has an eight-clock penalty for a branch mispredict).

Taken together, the rate and penalty represent a minimum two-clock penalty per branch or a four-clock penalty per iteration. This is a significant performance penalty for a seven-clock sequence; it is completely eliminated by predication.

Predication and control speculation have the potential to expose ILP and reduce the cycle count for general-purpose integer code. Although the two techniques are well known, IA-64 will be the first commercial architecture to incorporate both. This article gives the reader a glimpse of how IA-64 will be an important synthesis of these and other ideas in computer engineering. Intel intends to release more details in the coming months. ❖

References

1. S. Mahlke et al., "Characterizing the Impact of Predicated Execution on Branch Prediction," *Proc. Micro 27*, IEEE CS Press, Los Alamitos, Calif., 1994, pp. 217-227.
2. W. Hwu et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *J. Supercomputing*, Jan. 1993, pp. 229-249.

Further reading

Colwell, R., et al., "A VLIW Architecture for a Trace Scheduling Compiler," *Proc. 2nd Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, IEEE CS Press, Los Alamitos, Calif., 1987, pp. 180-192.

Kathail, V., M. Schlansker, and B.R. Rau, "HPL PlayDoh Architecture Specification: Version 1.0," Hewlett Packard Computer System Lab., Tech. Report HPL-93-80, Feb. 1994.

Rau, B.R. et al., "The Cydra 5 Departmental Supercomputer—Design Philosophies, Decisions, and Trade-Offs," *Computer*, Jan. 1989, pp. 12-35.

Schuette, M.A., and J.P. Shen, "Instruction-Level Experimental Evaluation of the Multiflow TRACE 14/300 VLIW Computer," *J. Supercomputing*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993, pp. 249-271.

Carole Dulong is a principal engineer and computer architect with the Microprocessor Products Group at Intel. Her research interests include compiler technology for ILP, architectures for 3D graphics, and multimedia applications. Dulong has a diplome d'ingenieur degree in from the Institut Superieur d'Electronique de Paris. She is a member of the IEEE and the IEEE Computer Society.

Contact the author at Intel Corp., 2200 Mission College Blvd., RN6-16, Santa Clara, CA 95052-8119; cdulong@mipos2.intel.com.